2011-03

Bandwidth and detection of packet length covert channels

Dye, Derek J.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/5724

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

BANDWIDTH AND DETECTION OF PACKET LENGTH COVERT CHANNELS

by

Derek J. Dye

March 2011

Thesis Co-Advisors:	George W. Dinolt James Bret Michael

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 25-03-2011	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* 2009-01-05—2011-03-25

4. TITLE AND SUBTITLE Bandwidth and Detection of Packet Length Covert Channels	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Derek J. Dye	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Navy	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A

14. ABSTRACT

This thesis explores the detectability and robustness of packet length covert channels. We discovered that packet length covert channels, where a rogue user modulates the length of a Transport Control Protocol packet, can be detected while monitoring traffic of a large network. The bandwidth of these channels can be successfully estimated as well as the channels themselves detected using statistical inference.
In addition, we observed that there is an inverse relationship between the volitionality in networks with respect to packet lengths and the detectability of these channels, and between packet length and channel bandwidth. For a large network like a college department, the bandwidth of a covert channel could be in the tens of megabytes over the course of a day.

15. SUBJECT TERMS

Covert Channels, Intrusion Detection Systems, Computer Security, Computer Networks, Operating Systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	79	

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 8–98)** **Prescribed by ANSI Std. Z39.18**

THIS PAGE INTENTIONALLY LEFT BLANK

BANDWIDTH AND DETECTION OF PACKET LENGTH COVERT CHANNELS

Derek J. Dye
Lieutenant, United States Navy
B.S., University of Maryland Baltimore County, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2011

Author:	Derek J. Dye
Approved by:	George W. Dinolt Thesis Co-Advisor
	James Bret Michael Thesis Co-Advisor
	Peter J. Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis explores the detectability and robustness of packet length covert channels. We discovered that packet length covert channels, where a rogue user modulates the length of a Transport Control Protocol packet, can be detected while monitoring traffic of a large network. The bandwidth of these channels can be successfully estimated as well as the channels themselves detected using statistical inference.

In addition, we observed that there is an inverse relationship between the volitionality in networks with respect to packet lengths and the detectability of these channels, and between packet length and channel bandwidth. For a large network like a college department, the bandwidth of a covert channel could be in the tens of megabytes over the course of a day.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Acknowledgements

I'd like the thank my wonderful family for their support during this. My time in Monterey was a wonderful opportunity to spend time with them after many years apart. My love goes out to my fiancée, Sachi, who has been immensely supportive of me and all the time I had to devote to this paper.

I'd also like to thank the outstanding support of Jason Cullum from ITACS and ITACS management at NPS. Mr. Cullum's invaluable support allowed for the immense amount of data collection to go forward.

Lastly, I'd like to thank my thesis advisors, Dr. George Dinolt and Dr. Bret Michael. You both supported my ideas and helped repeatedly with formulating them into high-quality research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

Cryptology predates digital computing by approximately 3000 years, but modern information systems offer opportunities and pose challenges to communicating secretly. It is not uncommon for users to want to keep even the fact that they are communicating secret. Keeping secret the fact that you are communicating is increasingly difficult due to layered network defenses in which traffic monitoring and analysis takes place. To avoid detection, users can employ covert channel techniques. The covertness of such channels results from modulating characteristics of information systems that were not intended for communications in order to transmit information.

1.1 Research Questions

From a detection standpoint, covert channels pose a challenge. Many types of covert channels implemented on top of Internet traffic exist but, in this thesis, we will treat only channels that modulate the packet length in network traffic. This thesis addresses the following questions:

How can a packet length covert channel be detected?

Some research has already been conducted on detecting packet length covert channels. However, these efforts are directed at individual connections. Can these channels be detected in large scale networks? Packet lengths in normal network traffic exhibit predictable patterns, with lots of traffic in the short and long lengths. In this thesis, we use statistical methods to attempt to detect packet length covert channels at a network-wide level.

What is the bandwidth of a packet length covert channel?

It is worthwhile for both the covert user and the network owner to know the bandwidth of a potential covert channel. From the covert user's perspective, it is good to know how much data can be transmitted, while from the administrator's perspective, it is important to know the data rate at which exfiltration can occur.

1.2 Thesis Roadmap

To start with, we will discuss previous research on covert channels. This covers covert classifications of curent channels and methods of detecting them. We outline some of the current research specifically related to packet length covert channels.

Next, we outline a packet length covert channel we developed by modifying the Linux kernel. This modification allows network packets being sent by a legitimate user to be modulated for the channel's purposes without any knowledge of the user. This modulation does not modify any of the data sent by the legitimate user.

Chapter 4 will describe the statistical techniques used to detect packet length covert channels. The techniques are general statistical methods using standard deviations and averaging. The methods are then tied together to also estimate the bandwidth of the covert channel. The next chapter outlines the framework for the experiments and testing of the equations from Chapter 4. The experiments use collected network traffic from an entire school department of several hundred users and use injected traffic to simulate a covert channel.

A large amount of network traffic was collected for analysis. Chapter 6 presents the analysis of all the data to determine how accurate Chapter 3's framework was. Lastly, we suggest future research to include how to improve the covert channel's robustness, as well as how to refine the equations introduced in Chapter 4.

CHAPTER 2:
Background

2.1 Introduction

In 1973, Butler Lampson characterized covert channels as unintended pathways for information transfer [1]. These channels typically use storage and timing characteristics of the system to bypass security measures and transmit information covertly. Although they tend to be low-bandwidth, covert channels provide adequate communication resources for exploiting and attacking computing systems, as evidenced by their use in rootkits and botnets [2].

2.2 Why Use Covert Channels?

Covert channels offer users methods to communicate and transmit information that is not often monitored by current Intrusion Detection Systems (IDS) or that is very difficult to detect. Users in need of this kind of secrecy include both State and non-State actors. These users need the confidentiality that encryption offers, but they also need something more. Their central need is to maintain the fact that they are communicating. It is easy to detect if a user is communicating even if the user is using encryption, especially if the detector is an administrator or has direct access to the communications medium.

A covert channel scenario can be illustrated by the Prisoner's Dilemma Problem introduced by Simmons in 1983 [3]. There are two prisoners in separate cells who want to talk to each other and have already agreed on the protocol of their covert communications. All messages must first be screened by the warden who either drops the message, forwards the message, or makes a modification before forwarding. The two prisoners then communicate by sending seemingly harmless messages on an overt channel that have hidden content encoded in them [3], [4]. For instance, the signaling could consist of varying the days that the prisoner sends a message or the number of words that he puts on a single page. One major factor in these scenarios is that the covert channel is dependent on an overt channel. The covert channel cannot exist otherwise, and its bandwidth is directly tied to the bandwidth of the overt channel. Figure 2.1 shows a model covert channel in which Alice is the sender and Bob is the receiver [5]. Alice modulates the overt message with the covert message and sends it. Bob then receives the message and extracts the covert message from the overt message. Figure 2.1 also shows how someone in the middle (Wendy) could modify the overt channel.

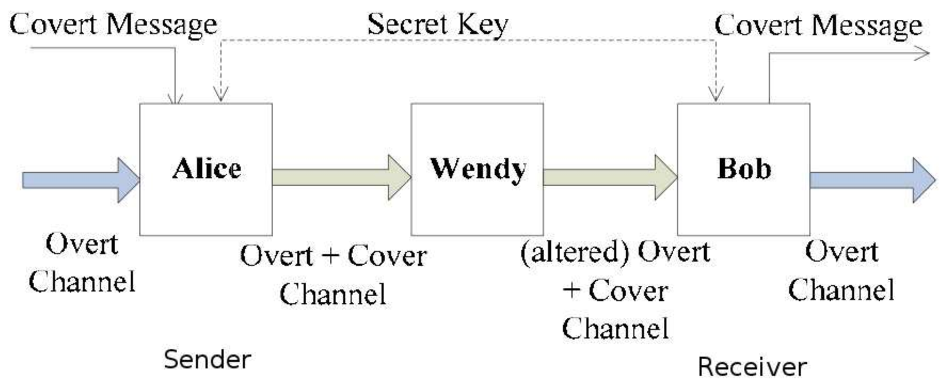


Figure 2.1: Covert Channel Model

2.3 High-Assurance Systems

Covert channels are of concern in high-assurance systems. Developers of such systems should attempt to identify and limit the bandwidth of such channels. In these systems, a covert channel is classified as either a:

Storage Channel: One process, the sender, directly or indirectly writes to a storage location while the second process directly or indirectly reads from the same location

Timing Channel: The sender modulates the timing of a system resource in a way that affects the response time as viewed by a second process

The Trusted Computer Security Evaluation Criteria (TCSEC) required analysis for storage channels and timing channels in order to meet B2 and above certifications [6].

An example of each type of channel is illustrated in the following scenario. Assume two processes are operating on a multilevel security (MLS) system with one at a high security level and the other operating at a low level. If the high process could modulate the intensity of its execution in such a way as to deny the processor to the low-level process, the high process could modulate the denial of the processor to transmit information to the low process. This assumes the system does not give equal time slots to each process or that the system does not give priority to the low process.

For storage channels, assume both processes have access to the same memory partition. First, the low process will try to write to the partition at regular intervals and will get an error from the system if the disk is full. Second, the high process fills up the partition and deletes from it. The low partition will get an error message when the high partition fills up the partition.

Three major methods exist to detect covert channels in high-assurance systems [7]:

Noninterference: The user analyzes process interactions to see if one process can interact or interfere with another process. If one process can interfere with another, then a covert channel exists. This is used to uncover primarily storage covert channels.

Shared Resource Matrix (SRM): Develop a matrix of all resources on one side and on the other all the processes. Then, determine which process uses which resource and whether it writes, reads, or modifies the resource. A covert channel can develop where two processes share certain properties on the same resource. This method can be used for identifying both timing and storage channels [8].

Information Flow Analysis: Analyze kernel system calls and variables that are accessible by users. The user must determine if two processes can view or alter any of these variables using exceptions or the system calls. This method is similar to that of the SRM.

Covert channels have also been created by modulating packet timing, data and headers of network traffic. With the proliferation of network protocols and network traffic since Lampson first defined covert channel, this is an area ripe for research. Gigabit LANs and broadband connections are becoming widespread. If a major server were to be compromised, an attacker would only need one bit per packet to covertly transmit upwards of 26GB of data per year [9].

2.4 Network-Based Covert Channels

A broader definition of covert channel has also emerged with network traffic. This broader definition includes any attempt to hide information in network protocols and refers to that information as hidden or covert information. This contrasts with methods like steganography, which refers to hiding information in content [4]. In this thesis, we use the broader definition of covert channel to refer to any manipulation of network timing or protocols to transmit information.

One of the simpler methods of creating such a covert channel relies on using unused header fields in network protocols like TCP/IP or ICMP. There are several fields in these protocols that do not have standardized default values, or that are not normally checked by network devices. For example, several covert channels use the Type of Service (TOS) field or several flag values in the TCP header [4], [10]. Both IPv4 and IPv6 have options fields that can be used if needed. It has been identified that an attacker could use the IPv6 Hop-by-Hop, routing, fragment, authentication and encapsulating security payload extension headers to embed covert information [4].

In addition to unused fields, commonly used fields can also be used. Two methods modulate the initial sequence number field of TCP. One takes one covert byte, multiples it by 256, and uses that for the TCP's Initial Sequence Number(ISN). Another method encrypts the covert data and uses that as the ISN. The second method results in uniformly distributed ISNs [4], [11].

Another common field to manipulate is the checksum field. The checksum field of an IP packet can be encoded with a secret message and an extra header extension can be added to ensure the IP's checksum matches the covert channel checksum. In UDP, the checksum is optional. In this case, an attacker could turn the checksum on or off to encode one bit of a covert message per packet [4].

A method similar to traditional timing attacks using shared processors is to modulate the time between sent packets. This works when the sender uses discrete time intervals between each packet sent. The number and length of discrete time values can vary from two to however many the receiver can distinguish one value from another. For a two-value system, each packet sent conveys one bit of the covert message. For multi-rate channels, the system can transmit $\log_2 r$ covert bits per packet, where r is the number of discrete timing values. This method does require some means to establish and maintain time synchronization [4], [12].

2.5 Countermeasures

2.5.1 Identification

Covert channels are meant to be difficult to detect, so detecting and eliminating them can be challenging. Because of the complex nature of computers and networks, in most cases it is not even possible to completely eliminate the potential for covert channels or to prove their non-existence. But, there is a set of methods that can be employed to lessen the potential or degrade their possible bandwidth [4]. With these methods, the user has to balance the danger of loss of information due to covert channels against the impact these methods have on performance and usability. In [4], Zander outlined a two-step process to develop countermeasures, with the first step being to identify the covert channel being used. The second step is to apply countermeasures, which he grouped into four broad areas as follows:

1. Eliminate the channel
2. Limit the bandwidth of the channel
3. Audit the channel
4. Document the channel

In traditional high-assurance systems, identification takes the form of semi-formal methods. Some of the methods include information flow analysis, noninterference analysis, SRM method, and the covert flow tree method [4]. These methods can be used during the design phase of a system.

Less work has been done on the formal analysis needed to identify and measure the capacity of covert channels in network protocols. Donaldson suggested that the SRM could be applied for this purpose by splitting the host-to-host channels from the intra-host channels on a single computer [4], [13]. Helouet proposed a requirements-level analysis for covert channels on distributed systems [4].

Countermeasures can be applied once a covert channel has been identified. In some cases, the channel is available due to a mistake made in building the system and can usually be fixed with a patch or similar fix. More complex methods need to be applied if the channel is due to systemic aspects of the system or due to unwillingness to reduce functionality [4].

Zander and others believe that in real-world situations it is impossible to eliminate all covert channels. Hence, we should eliminate the ones we can and apply the remediation methods to the ones that remain [4].

2.5.2 Eliminating Channels

The first line of defense is to have secure systems. This could include running hardened kernels, regularly installing software and firmware updates, and ensuring services are properly configured. This will help mitigate the spread of viruses, Trojan horses, rootkits and other software that could insert and exploit a covert channel. A secure system should also limit the number of services to only the ones that are required. For example, if a web server is not needed, do not allow it to run. Also, ICMP is not needed by most users and can be disabled [4].

Many of the network headers manipulating covert channels can be eliminated by normalizing header fields. This can take the form of strictly enforcing protocol specifications or where they are not well defined, setting the values to zero. Zander gives an example with the Don't Fragment (DF) bit. If the DF bit is not set, a normal system probably would not check the IP Identification number, which would be needed if an IP packet was fragmented. Setting the IP ID number to zero every time the DF bit was zero would eliminate use of the IP ID number as a covert channel [4]. Secure systems should also apply this technique to optional header fields that should be all zeros if not in use. Handley outlines a broad collection of methods to normalize network header fields [14]. These methods add some extra overhead, which would have to be balanced with existing requirements.

2.5.3 Limiting Capacity

Information about a channel's bandwidth is useful for applying remediation methods. With some channels, it is easy to determine the bits-per-packet capacity. For example, if a covert channel is modulating a single bit, the per-packet capacity is one bit. The bits-per-second rate is harder to determine, since it is dependent on the overt traffic on which the covert channel depends.

If a channel is sensitive to noise, such as timing channels, a server could intermittently send dummy packets to other systems to try and add some noise to the channel [4].

Packet-length-based covert channels can be limited by reducing the possible lengths a packet could have thereby limiting the number of states a packet-length covert channel could use. If a packet is too small, the packet could be padded with extra zeros. The disadvantage to this approach is the wasted bandwidth. If bandwidth were not a concern, a user could completely remove this type of channel by allowing only one possible packet length on the network [15]. Limiting timing channels by varying the time between packets can limit the capacity of a channel. For example, one high-assurance system uses a store-and-forward router to transfer information from low to high. In this system, the low sends data to the high while the high sends acknowledgments to the low. To prevent the high system from using the acknowledgments as a timing covert channel, the router could store the acknowledgments and only transmit them after a small random amount of time. This does not completely eliminate this timing channel, but it does reduce its bandwidth [4], [15].

2.5.4 Detection Methods

In real-world situations, it is going to be too expensive or technically impractical to secure every system on a network. As a result, methods have been developed to test for covert channels. Most of these methods rely on the fact that the behavior of the covert channel differs measurably from normal traffic. An example is the header-based channels like initial IP ID and TCP ISN covert channels that result in random values. Normal IP ID and TCP ISN numbers actually follow a fairly predictable distribution, with the actual pattern tied to the operating system. It is possible to detect changes from the anticipated patterns with a high degree of certainty. The probability of detection drops significantly once the covert channel has a distribution close to that of normal traffic [4], [16].

A similar method can be used for timing based covert channels. Venkatramn proposed measuring the average time between packets for a given network. If the rate increased or decreased for a host by a certain trigger amount, this would indicate a covert channel [17]. This method can produce an increasing rate of false alarms and missed detections the further the normal traffic deviates from the average.

2.6 Packet Length-Based Covert Channels

From this point forward in the thesis, we treat only packet length covert channels. There have been several reported research investigations into packet length-based covert channels. Two papers highlighted in this section, Liping [5] and Nair [18], look at covert channels over a UDP packet network. The papers are narrow in scope and do not address detection of covert channels from a network-wide perspective.

2.6.1 Implementation of Packet Length-Based Covert Channels

This subsection provides an overview of Liping's paper, entitled "A Novel Covert Channel Based on Length of Messages" [5].

This paper builds on earlier work by attempting to make a packet length covert channel have a distribution of packet lengths similar to normal traffic. The authors claim that their approach is protocol-independent and flexible enough to markedly lower the probability of detection.

Their method uses samples of network traffic to build what they call a reference. This reference is used to determine what length to use and adjusts future packet lengths to ensure a normal distribution. The steps of transmission are as follows:

1. Sender and receiver communicate normally. Both record the packet lengths sent and record the numbers in the reference.

2. Sender and receiver select a length l from the reference using the same agreed upon algorithm.

3. Using an agreed-upon packet number, the sender sends a message of length l plus a number based on the contents of the packet. The reference is then updated with the new length.

4. Receiver determines the message by subtracting the number based on the contents of the packet from the packet length.

5. Steps 2 to 4 are repeated until the entire message is sent.

The paper then outlines their experimental results and how they compare to other covert channel schemes that have random distributions. From their results, it appears that their method matches a normal distribution more closely than the other methods.

The Liping paper does not provide any detail on how the reference is constructed from normal traffic nor does it detail how it is supposed to allow the covert channel to mimic normal traffic. This would make it difficult to replicate the results or to rely on the findings of the study. While the research shows promise at normalizing covert channel traffic, the paper lacked enough supporting information to make a convincing case that their method avoids detection by mimicking normal traffic.

2.6.2 Possible Method to Detect Packet Length Covert Channel

This subsection gives an overview of Nair's paper, entitled "Detection of Packet Length Based Network Steganograpy" [18].

Nair investigated how to detect the covert channel introduced by Liping. The author developed a mathematical detection mechanism. The claim is that the mechanism can detect Liping's channel in addition to detecting their varient of Liping's covert channel.

The author developed a testbed centered around a UDP protocol chat program implemented in Java. Nair then ran the program using chat data to determine what the normal distribution looks like. For their detection method, Nair introduced a function they call a packet-length vector. It is a first-order statistic using the number of packets for each of the valid packet lengths representing the relative frequency of a particular packet length in the series of packets.

Nair then used a technique that has been applied in steganography of images to compare the normal packet length vector to a covert channel vector. Nair found that changes in packet lengths will result in changes in the packet length vector that make the graphed vector less smooth. This change can be quantified by first performing a discrete Fourier transfom (DFT) on both vectors. Then calculate the center of mass (COM) of the vector according to the following:

$$COM = \frac{\sum_{i=0}^{N-1} iX_{dft_i}}{\sum_{i=0}^{N-1} X_{dft_i}} \tag{2.1}$$

Here X is the packet length vector, while X_{dft} is the DFT performed on X. The summation adds each element in the DFT vector. Nair observed that $COM(Covert)$ is usually greater than $COM(Normal)$, where covert is the covert channel DFT and Normal is the normal traffic DFT.

Nair's implementation of Liping's covert channel is susceptible to detection simply by analyzing the graph of packet length relative to time. Nair modified Liping's algorithm to be less susceptible to detection, but did not specify what was changed.

Nair's results are promising, providing for detecting covert channels especially their application of DFT's to network-based covert channels. However, Nair's data was limited to a simple standalone UDP-based chat program. It is unknown whether Nair's method could detect other types of packet length-based channels on other protocols or services and whether that traffic could be detected when mixed with other traffic.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Covert Channel Design

This chapter describes a packet length-based covert channel. We will refer back to this type of channel later in the thesis. The channel might be part of a rootkit covert-communications capability. Once installed, the channel could be used to communicate between two hosts, but only if a sufficient level of overt traffic is going between the two systems to minimize the probability of detection of the channel's transmissions.

3.1 Channel Structure

The covert channel relies on assigning a series of symbols or values to separate packet lengths, which we call the reference. The reference is shared by the sender and receiver by some channel separate from the covert channel. This could be distributed by any number of means, which might include hard-coded in the rootkit, embedded in an image on a well-known website, or embedded in an e-mail. The channel can use as many or as few packet lengths as desired, depending on the needs of the participants. If viewed in terms of information theory, the packet lengths are the symbols of the channel and, as long as the symbol transmission remains constant, one can increase the bandwidth of the channel by increasing the number of packet lengths used [19].

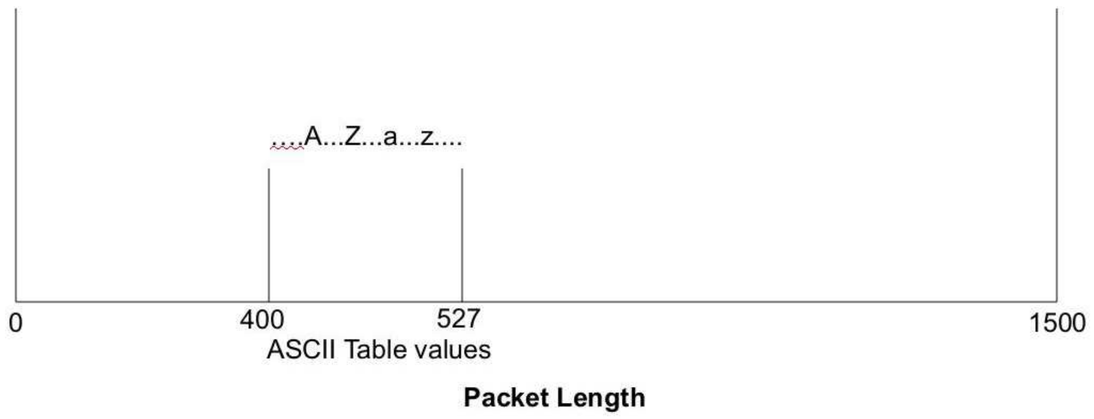


Figure 3.1: Covert Channel Structure

In our case, we assign a packet length to each character in the ASCII table. We use consecutive packet lengths between 400 and 527, but this is an arbitrary range since the channel can be in non-consecutive packet lengths of any length spread across the possible packet lengths of the overt channel. Figure 3.1 shows the possible packet lengths of an Ethernet 1500 bit frame.

To simplify implementation, we assume that any packets arriving from a known covert channel peer that have a packet length between 400 and 527 are a part of the covert channel. If no covert channel traffic is to be sent, no packets would have a length within that range. Lastly, the user manually sets the address of the remote user they wish to communicate with. This could also be hard-coded for ease of use.

3.2 Implementation

We decided to implement the channel in Linux (kernel 2.6.31) since the system is open source and allows for easy modifications. One assumption is that the overt users have no idea that their traffic is being modified for use in a covert channel. To accomplish this, the easiest way is to modify the kernel itself instead of relying on modifying applications like the browser Firefox. In order to do this, we modified the network stack of the kernel. This allowed us access to any traffic being sent out, regardless of what application was generating it. We modified the kernel's network stack code directly. This could also be accomplished in a kernel module that can be separately installed. This would allow a kernel module rootkit to easily modify a running unmodified kernel.

The overall structure of the channel is a modified TCP stack inside the kernel. The modified code maintains send and receive buffers inside kernel-space. All users interact with the channel via new system calls. This allows a program to be written in user-space to use the channel without having to modify the kernel.

Linux's network stack is highly modularized with separate function definitions for each protocol. We chose to implement the channel with TCP, so we concentrated on those functions. We modified tcp.c, tcp_output.c, and tcp_input.c found in the net/ipv4 directory. The kernel source code location depends on the distribution, but usually can be found in the /usr/src/linux directory. See Appendix A for a complete listing of the source code.

In tcp.c, we set up the infrastructure to allow a user to communicate using the covert channel. New send and receive buffers were set up inside the kernel along with buffer index variables. New variables were also set up to save the IP address of the covert channel peer. In the case where there are multiple TCP connections between two peers, the covert channel only uses one of those connections. This simplification removes the problem of having the covert channel reassemble the covert message spread across multiple connections. This is enforced within the kernel using a Boolean int variable (is_cov_channel) added to struct sock_common found in sock.h and another Boolean int variable in the kernel (cov_channel_locked).

To change the packet length, the Maximum Segment Size (MSS) is modulated. This forces the rest of the kernel code to shorten the packet to the desired packet length, creating the desired effect of modulating the packet length. The MSS value is set to the maximum size a packet can be without getting fragmented. This value rarely changes, and is normally 1460 for regular Ethernet. See Appendix A.3, starting at line 61, or the discussion below for details.

Three new system calls were added to allow processes to use the covert channel within user-space. One system call allows a user to add a message to the send buffer, one allows a user to read from the receive buffer, and another allows a user to set which peer to communicate with. Lastly, four non-system call helper functions were added for use within the kernel. If the user does not require user-space functionality, these functions can be removed with the result that all interaction with the channel is restricted to kernel-space.

3.2.1 Transmitting

After a TCP connection is established in Linux, the tcp_write_xmit function in tcp_output.c is called every time data is sent. For the covert channel, it first must test if the channel is being used by any other connection (Listing 3.1). The connection executing tcp_write_xmit will be used and locked for the covert channel if the channel is not already locked by another connection $(!cov\_channel\_locked)$ and the destination address for the connection is equal to the covert channel peer address $(inet->daddr == cov\_channel\_address)$.

Listing 3.1: Locking a TCP connection within tcp_output.c

```
if(!sk->is_cov_channel && !cov_channel_locked
   && inet->daddr==cov_channel_address) {
          cov_channel_locked=1;
          sk->is_cov_channel=1;
}
```

Once the channel is locked, the user can use the connection to send data over the covert channel. Next, the connection checks if there are any covert messages to be sent $(sk->is\_cov\_channel$

&& $packet! = -1$), as well as if there are any overt messages of sufficient length $(skb- > len\,(400 + packet))$. If those two conditions are true, the packet will be processed for the covert message (Listing 3.2). The connection reads one character from the send buffer and adds its integer value to 400. For example, assume the character read from the buffer was “d.” The ASCII value for “d” is 100, so the MSS value would be $400 + 100 = 500$. If the overt message has a length that falls inside the covert channel, then the kernel will reduce the MSS to 399, which is right below the covert channel. For other cases, the MSS value will be set using its normal size.

Listing 3.2: Decision Point within tcp_output.c

```
packet=channel_packet_check();
int packet_used=0;

if(sk->is_cov_channel && packet!=-1 && skb->len > (400+packet)) {
        packet_used=1;
        printk(KERN_EMERG "package_to_send␣-%c␣-%d",packet,packet);
        mss_now = 400 + packet;
}
else if(skb->len >= 400 && skb->len <=527)
        mss_now=399;
else
        mss_now= real_mss;
```

If the message was successfully sent, the connection will remove the packet from the covert buffer by calling channel_get_package_to_send(). See Listing 3.3.

Listing 3.3: Transmitting in tcp_output.c

```
        if(packet_used)
                packet=channel_get_package_to_send();
        if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
                break;
```

The covert channel code adds some additional overhead as compared to unmodified Linux, but it should be a small impact. It does add a handful of conditional statements into every packet being sent, and another handful of array operations if a covert channel message is being sent. Given that unmodified Linux has many more conditional statements associated with each packet sent, this should be a small addition. This thesis leaves performance and quality of service issues to future research.

3.2.2 Receiving

Receiving requires much less redesign to implement. The only modification was made to tcp_event_data_recv function found in tcp_input.c, which is called every time an TCP packet is received. The function begins by first checking whether the TCP packet originated from the address of a covert channel peer and also if the packet length is in range of the covert channel. See Listing 3.4. If those cases are true, the connection subtracts 400 from the packet length to derive the original covert packet. It then adds it to the receive buffer by calling channel_put_package_received().

Listing 3.4: Receiving tcp_input.c

```
if(inet_sk(sk)->daddr==cov_channel_address
    && skb->len >= 400 && skb->len  <=527)
            channel_put_package_received(skb->len -400);
```

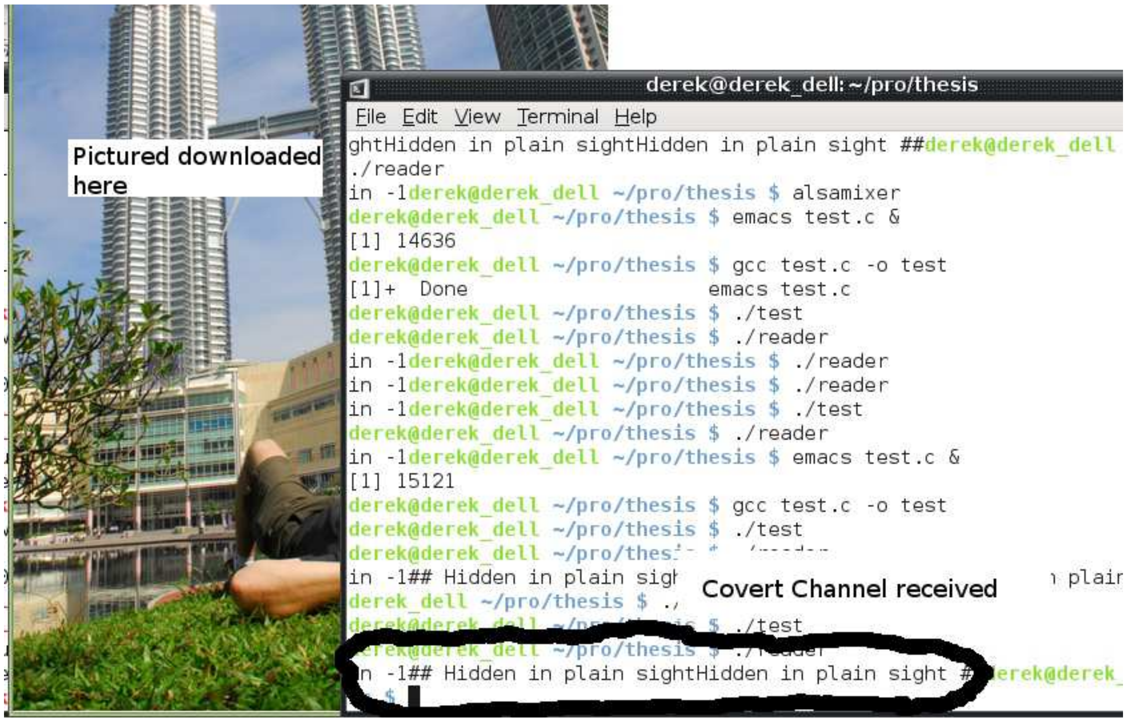
3.2.3 Code Demonstration

The user interacts with the covert channel using three new system calls. These calls maintain the send and receive buffers, which are simply statically defined integer arrays of size 1000. The send buffer is a circular buffer, while the receive buffer is a first-in-first-out (FIFO) buffer.

We wrote a C program that used the system calls to demonstrate the channel (Appendix A.4). The program has three wrapper functions that directly use the system calls by putting strings into the send buffer and by reading from the receive buffer.

The testing was conducted on a local area network with Internet access that had a web server on one computer and the example program on a client machine. The website was a series of pictures hosted on an Apache server running Linux. The kernels on both machines were modified with the covert channel. The test program was run on the website server and inserted the text “Hidden in plain sight” onto the send buffer. A user on the other machine downloaded a picture from the website and then ran the test program to read from the buffer. Figure 3.2 shows the image that was downloaded and the print output of the test program showing the covert channel message.

Later testing had the user view multiple websites on other machines before and after downloading an image from the covert channel website. During the testing, a network dump was conducted using wireshark. Figure 3.3 gives a visual snapshot of packet lengths from 300 to 1425 from the network dump. The horizontal axis is the packet number, which roughly corresponds with time while the vertical axis is packet length. The covert channel can easily be spotted as the compact groupings of packets between 400 and 600 packet lengths in the bottom left of the image. Most of the remaining packets are legitimate traffic such as HTTP page requests.



```
ghtHidden in plain sightHidden in plain sight ##derek@derek_dell
./reader
in -1derek@derek_dell ~/pro/thesis $ alsamixer
derek@derek_dell ~/pro/thesis $ emacs test.c &
[1] 14636
derek@derek_dell ~/pro/thesis $ gcc test.c -o test
[1]+  Done                    emacs test.c
derek@derek_dell ~/pro/thesis $ ./test
derek@derek_dell ~/pro/thesis $ ./reader
in -1derek@derek_dell ~/pro/thesis $ ./reader
in -1derek@derek_dell ~/pro/thesis $ ./reader
in -1derek@derek_dell ~/pro/thesis $ ./test
derek@derek_dell ~/pro/thesis $ ./reader
in -1derek@derek_dell ~/pro/thesis $ emacs test.c &
[1] 15121
derek@derek_dell ~/pro/thesis $ gcc test.c -o test
derek@derek_dell ~/pro/thesis $ ./test
derek@derek_dell ~/pro/thes
in -1## Hidden in plain sigh                              plair
derek_dell ~/pro/thesis $ ./
derek@derek_dell ~/pro/thesis $ ./test
derek@derek_dell ~/pro/thesis $ ./reader
in -1## Hidden in plain sightHidden in plain sight ##derek@derek_
 $
```

Figure 3.2: Example Screenshot of Program Execution

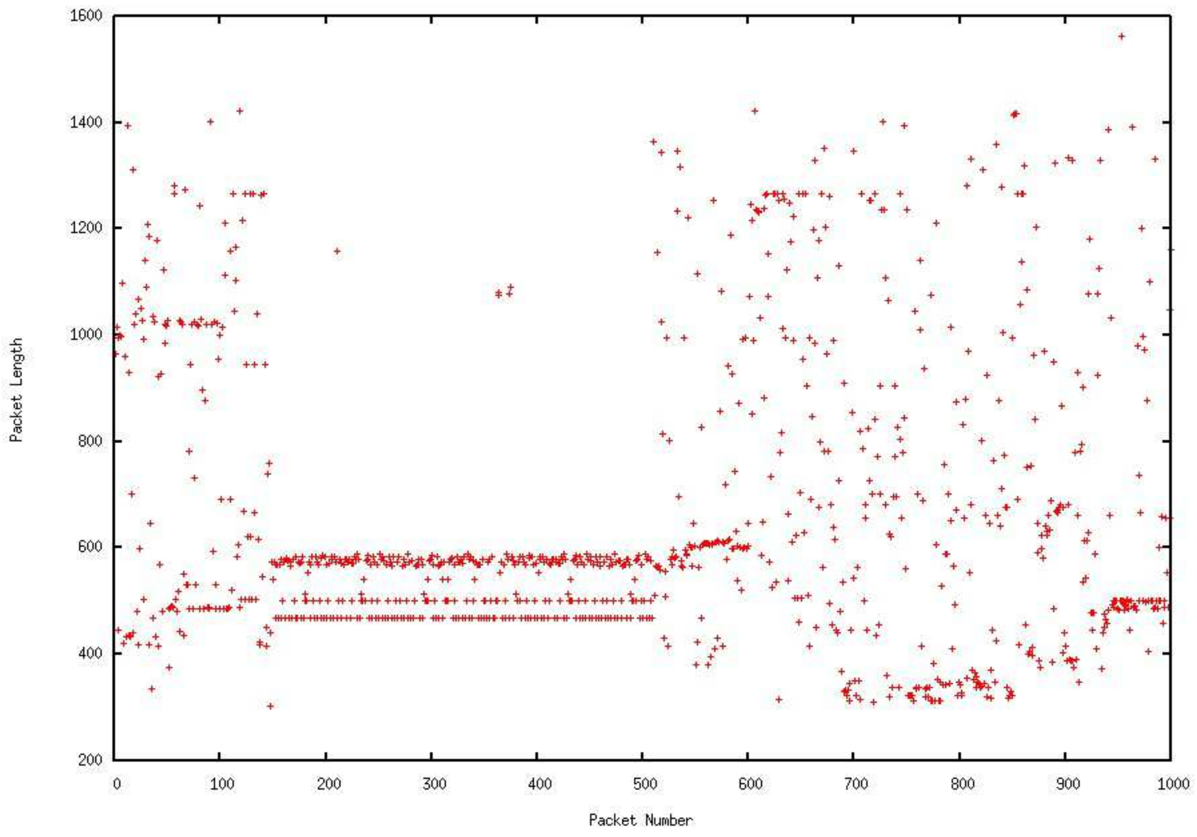


Figure 3.3: Sample Network Traffic Displaying Packet Number and Packet Length

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Statistical Approach to Detection and Bandwidth Analysis

Both references [5] and [18] treat the detection of a packet-based covert channel in the context of a single connection. In this thesis, we are interested in how an IDS can be used to detect a packet length-based covert channel and what bandwidth that channel could attain while escaping detection. The IDS would be used to monitor an intranet or on a network gateway. This chapter outlines the framework that can be used to estimate the bandwidth of the channel, as well as a system that can be used by the IDS.

4.1 Maximum Bandwidth

Covert channels always require overt information to transmit data, so the bandwidth of the covert channel will be tied to the bandwidth of the overt channel. In the case of the covert channel from Chapter 3, the covert packet is dependent on the size of the overt channel packet. This packet size or segment size is normally 1460 bytes for Ethernet. So, for every segment size section of overt channel bandwidth, the covert channel can encode some information. The amount of information depends on how many packet-lengths or channels are being used for the covert channel.

Converting the number of packet-lengths into bits can be done by taking the log base 2 of the number of packet-lengths in the covert channel or $log_2(channels)$. Therefore, each packet of the overt channel can encode $log_2(channels)$ of covert channel bits. The maximum bandwidth the covert channel can attain then becomes:

$$bandwidth_{covert} = \frac{\log_2(channels)\, bandwidth_{overt}}{segment\_size} \tag{4.1}$$

4.2 Detection Threshold

If a user used the maximum covert bandwidth, this should greatly alter the distribution of packet-lengths for the network. On the other hand, if there is a lot of traffic on the network and the user only sends a small amount of covert data, the change in the packet-length distribution would go undetected. Between these extremes is a threshold at which the covert channel becomes detectable.

Detection of the channel will depend on the characteristics of the overt channel. The distribution of packet lengths on a normal network has some predictable characteristics. There is a significant amount of traffic from 52 to 300 bytes, with much of this being TCP acknowledgment packets. There is also a large amount of traffic between 1300 and 1500 bytes. This is where much of the data is being transmitted. In between 300 and 1300 is much less data, but activity remains, with much of the HTTP page requests occurring here. The frequency and number of packet-lengths of the network will fluctuate, depending on many factors, such as the network applications being used, time of day, and day of week. Figure 4.1 shows a graph of a network's packet lengths over the course of a day. The hour scale goes from 0 to 24 hours, while the packet length goes from 50 to 1500. The Frequency axis depicts the number of times a given packet length occurred during each hour. The method for collecting this data is outlined in Chapter 5.

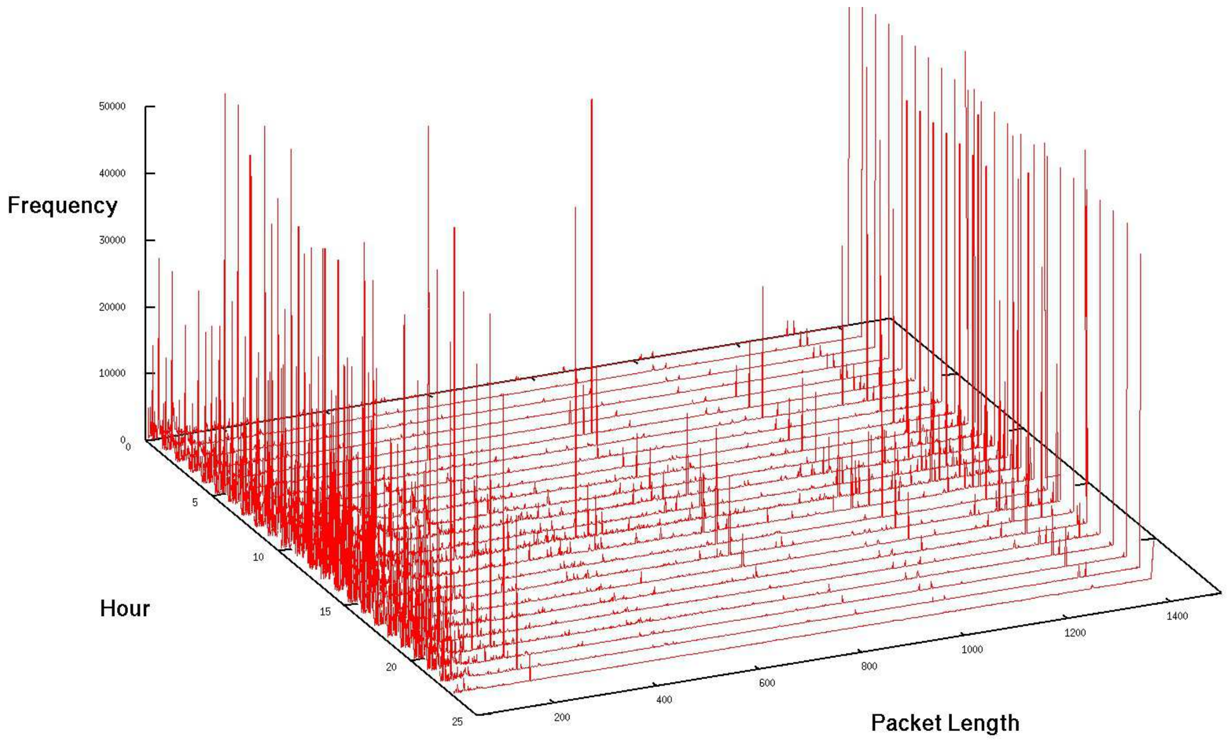


Figure 4.1: Single Day Graph of Packet Lengths from the School Network

We can construct a function $T(l)$ that will be called the threshold function. The function returns the minimum number of packets that can be sent with packet-length l that will be considered abnormal. Here we use statistical analysis for this function. Another approach could use DFTs.

In order for a covert channel to escape detection, it will have to transmit below the threshold set in $T(l)$. This will depend on the size of the covert message being sent and the frequency of the bytes (symbols) associated with length l. Let M be the covert message being sent in a given time frame and let q_i be the probability of symbol i appearing in M. Then, avoiding detection can be represented as:

$$Mq_i < T(l_i) \tag{4.2}$$

Several observations can be seen right away. For the channel to remain undetected, the transmitted data must remain in the noise by doing one of three things:

- Reduce the signal by reducing the size of M
- Spread the signal out by reducing the probability q_i. If the channel is encrypted, creating an evenly distributed cipher stream, q_i simply becomes $\frac{1}{channels}$. In this case, increasing the number of channels will decrease the probability.
- Pick a packet length with a greater threshold.

Summing all the thresholds for each packet length in the covert channel and subtracting normal traffic (N_i) should result in the total symbols of the channel. To convert it to bits, one can multiply by $log_2(channels)$ to get the maximum undetected bandwidth:

$$bandwidth_{covert} = log_2(channels) \sum_{i=1}^{channels} (T(l_i) - N_i) \tag{4.3}$$

4.3 Defining the Threshold Function

The threshold function can use any method that can distinguish the covert channel traffic from normal traffic. The goal is a function that extracts the covert channel signal from the noise of regular traffic. We will use statistical inference to compare new traffic with an historical average. The average data will be broken down by packet length and a user-defined unit of time such as hour of the day.

The function is:

$$T(l) = Cs_l + A_l \tag{4.4}$$

where l is the packet length of interest and s_l is the standard deviation of occurrences for l. A_l represents the average occurrences of l. C is a constant and can be used to adjust the number of standard of deviations one wishes to use.

4.4 Intrusion Detection Framework

In addition to using this framework to estimate the bandwidth of the covert channel, one can also use it to build a framework for intrusion detection. One can define a function that tracks the number of packet lengths that exceed the threshold function and then flag the traffic if it exceeds a threshold. This can be represented as follows:

$$E(l) = \begin{cases} 1 & \text{if} f_l \geq T(l) \\ 0 & \text{else} \end{cases} \tag{4.5}$$

$$\sum_{i=1}^{segment\_size} E(i) >= alert \tag{4.6}$$

Here, $E(l)$ returns 1 if the sampled frequency (f_l) for packet length l exceeded the threshold funciton, $T(l)$. A 1 represents a single alert. All other cases will return 0. The next equation states an intrusion has occurred if the sum of all alerts over the span of possible packet lengths has exceeded the user-defined cutoff $alert$.

This system will allow for estimating a covert channel's bandwidth, as well as for a system to detect these channels. Much of this follows signals theory and the cat and mouse game between detecting a signal among the noise and trying to hide that signal in the noise [20].

CHAPTER 5: Experiment

To test the system of equations outlined in Chapter 4, we devised a testbed that includes data obtained by sampling network traffic and injecting test traffic. Our network of choice was live traffic from the Naval Postgraduate School's Glasgow Hall, home of the NPS Computer Science Department.

5.1 Network Architecture

The school network is partitioned into several subnets. The Computer Science Department is part of the Graduate School of Operational and Information Sciences, which resides in Glasgow Hall East Wing. We conducted all our collection using a collector that sniffed traffic near the gateway to the Internet. See Figure 5.1 for a network diagram. We restricted all analysis to traffic coming and going from Glasgow Hall to the Internet and saved just the time the packet was received and the packet length.

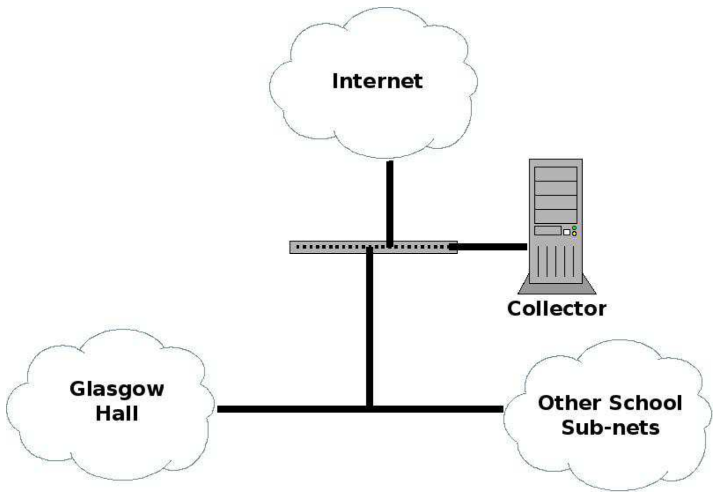


Figure 5.1: Network Diagram

5.2 Collection

The equations found in Chapter 4 require a baseline dataset to compare live traffic. One could use a dataset from an outside source, or the better option would be to sample one's own network.

To sample traffic on the school's network required several steps to get the collected data into a form we could analyze. The steps are shown in Figure 5.2.

1. Save network data in a standard form, conduct data reduction, and transfer to the computing platform upon which analysis will be conducted
2. Import data reduced files into database
3. Summarize raw data into units of time to be used for analysis (rawdata to count)
4. Conduct steps 1–3 until desired level of sampling has been attained
5. Calculate average and standard deviation on data set (count to ave)

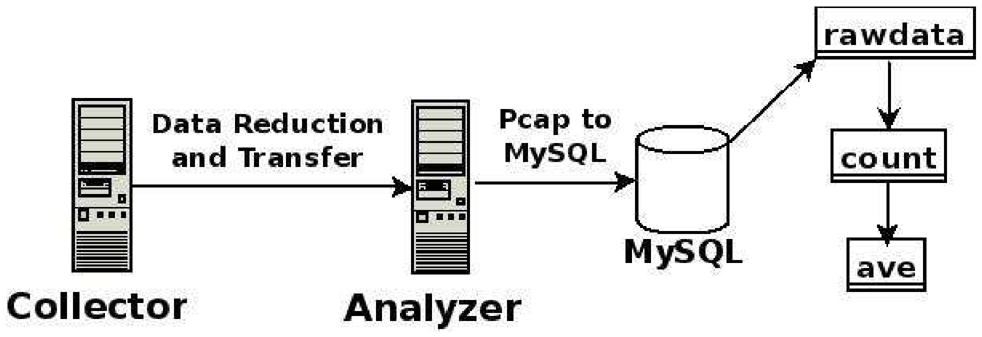


Figure 5.2: Transfer Process

Step 1 began by having the collector save the data in a standard way. The collector we used was a regular part of the network, and it saved all the packets that traversed the school network to the Internet. Entire packets in pcap format were saved in 650-megabyte files. Pcap format is a standard format for saving raw network packet information, and it is used by programs such as tcpdump and Wireshark [21].

Given the large amount of data generated by the collector, the files had to be reduced before sending them to the analyzer. The code outlined in Appendix B.1 was used to read the pcap files, strip off the Unix timestamp and the IP packet length, and save them to a separate file. To minimize the file sizes, only IP packets of sizes between 56 and 1499 were saved. The program also checked each packet to ensure it was part of the Glasgow Hall subnet. It did this by conducting a bit-wise "and" between the destination and source addresses and comparing it with the netmask of Glasgow Hall. Using a bit-wise "and" simply ands the bits of the address, which is a fast and simple operation. If either source or destination addresses matched the netmask, the packet information was saved. Another method used for step 1 was a custom program that read network traffic directly using the libpcap library and only wrote the UNIX timestamp and packet length to a separate file. The code for this program can be found in Appendix B.2.

Once each pcap file was reduced to the timestamp and packet length, the files were compressed and transfered to the computer upon which the data was analyzed. The files were then inserted into a MySQL database (MySQL v5.1.52 on Linux) using a bash script (Appendix B.3). The database was broken into three tables: $rawdata$, $count$, and ave. See Appendix C for the full SQL table definitions. $Rawdata$ simply saves all the data from each of the filtered pcap files into the database. Table $count$ sums the packet lengths from $rawdata$ for a unit of time that corresponds with the units used for the equations found in Chapter 4. The SQL code to compute $count$ is as follows:

Listing 5.1: Populating Table count using Table rawdata

```
insert into count select (timestamp>>10)<<10 as time, len, 1
        from rawdata on duplicate key update counter=counter+1
```

This command first takes everything from the table $rawtable$ using the select command. It then strips off the first 10 bits of the timestamp using bit-wise shift command to convert the time to 17-minute intervals. Unix timestamps are measured in seconds, with 10 bits approximately equaling 17 minutes. One can use another value instead of 10 to increase or decrease the unit of time, or one could use a modulo command to use a base other than 2. The bit-wise method was used since it performs faster than the modulo command. Lastly, the analyzer uses the duplicate key update command to sum the traffic.

Hourly averages are also computed, since there is a large variation in traffic based from hour to hour. For example, using the time interval of 17 minutes, a 17-minute average is computed for each hour of the day. The table ave is used to store the averages, which stores the hour, packet length, average, and standard deviation. The full definition can be found in Appendix C. To compute it, the following code was run:

Listing 5.2: Populating Table ave using Table count

```
insert into ave select len, avg(counter), stddev_pop(counter),
        hour(from_unixtime(timestamp)) from totcount
        group by len, hour(from_unixtime(timestamp))
```

5.3 Injection and Analysis Framework

Once the baseline dataset is obtained, one can begin sampling and analyzing the data. The analysis was done by comparing new collection against the table $average$ established previously. We conducted the analysis on an entire day's worth of sampled data, since it was easier on the collector to forward data on a daily basis. Further work could be done to increase the frequency with which analyses are run. This would speed up detection of a potential covert channel.

5.3.1 Analysis

The steps for analysis followed the same steps 1 through 3 of collection. We created a table for each of the analysis days with the same table properties as $count$ and $rawdata$. We did this to maintain all the data collected, but one does not need to do this. One could simply reuse the tables $rawdata$ and $count$ and simply delete the old data.

The following SQL command is used to compare the new data with the average using the equations from Chapter 4:

Listing 5.3: Comparing new data with baseline

```
select timestamp, hour(from_unixtime(timestamp)),
       count(c.len) from count as c, ave
       where  hour(from_unixtime(timestamp))=hour and
       c.len=ave.len and c.counter > (stdev*6+ave)
       group by timestamp
       order by hour(from_unixtime(timestamp));
```

First, the command computes which packet lengths within a single timestamp block (17-minute block) have exceeded $stdev * 6 + ave$ (Equation 4.4). In this example, we use 6 standard deviations. Next, the command counts the number of packet lengths that have exceeded the threshold for a given timestamp as outlined in Equations 4.5 and 4.6. This is done by using the count SQL operation in conjunction with the group by timestamp.

Another analysis tool graphs the traffic visually. This can be done using a SQL output like this:

Listing 5.4: Outputting count table for display in gnuplot

```
select hour(from_unixtime(timestamp)), len,
       sum(counter) from countjan06 group by hour(
       from_unixtime(timestamp)), len
       order by hour(from_unixtime(timestamp)), len
```

The output of this can then be displayed using a three-dimensional rendering program such as gnuplot [22]. This will create images similar to Figure 4.1. This form of analysis made it visually possible to determine if there was unusual traffic over the course of the day. Often, it could be determined quickly, using this method, that automated applications were running during the night because they generated large spikes throughout the night.

5.3.2 Injection

For the injection, it was decided to use arbitrary packet injection instead of using the actual covert channel code outlined in Chapter 3. This provided increased flexibility in choosing the packet length and frequency of injection. Injection was provided using a program called hping [23]. This program extends the capabilities of the basic ping program to include the creation on arbitrary packets of variable size using several networking protocols such as IP, UDP, and TCP. For the experiments, we used a machine within the Glasgow subnet to send packets of specific lengths to arbitrary sites on the Internet.

Testing followed these steps:

1. Inject a packet with specific length as many times as desired and log transmission.
2. Repeat step 1 with different packet lengths until desired.
3. Wait until unit of time as defined in Section 4.3 has elapsed. In this case, this was 17 minutes.
4. Repeat 1–3 until testing complete.
5. Process and analyze collected data.

Steps 1 and 2 were done in parallel using bash scripting.

The system of collection, analysis, and injection outlined in this chapter allowed for accurate testing of the equations and framework outlined in Chapter 4.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6: Analysis of Results

Several questions have been proposed on whether it is possible to detect packet length covert channels from a network wide perspective. To test this, three weeks worth of collection was performed on the school network, to include over a week of baseline collection. The baseline data provided the information to calculate the threshold functions in Chapter 3.

6.1 Summary of Baseline Data

The baseline data consisted of over 400 million packets collected over the course of several weeks during the work week (i.e., Monday-Friday).

The school's network traffic was found to be variable depending on time of day and day of the week. Figure 6.1 shows the daily packet totals for each day. Saturday and Sunday are also shown for reference. Figure 6.2 shows a graph of hourly traffic averages. The large spike in the late morning corresponded to when most of the computer science classes are taught.

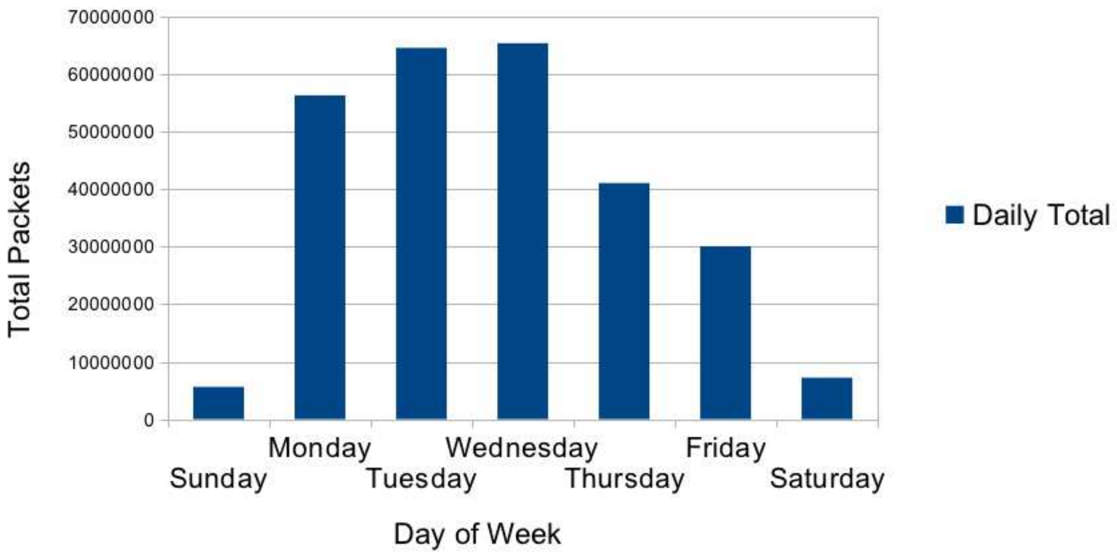


Figure 6.1: Daily Packet Totals

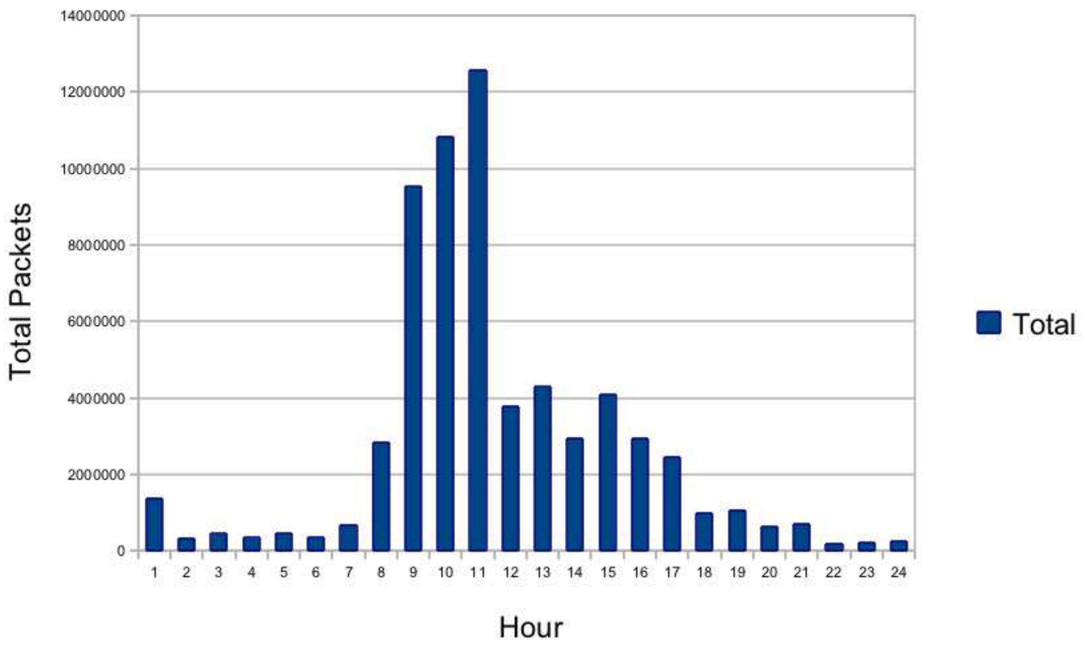


Figure 6.2: Hourly Packet Profile

It was also noticed that several applications ran over the course of days or ran during off-hour times. This had the effect of creating noticeable traffic during the middle of the night. This adversely affected detection by creating false positives if the baseline data had not seen the application run.

6.2 Detecting Packet Length Covert Channels: Narrow Scope

Before we could detect a covert channel, we needed to determine a good multiple for the standard deviation for Equation 4.4. We concentrated on packet lengths between 1181 and 1190. This area had low variability and low enough packet counts to make testing easier. Using the baseline data, it was determined that a multiple of 6 removed all false positives from normal traffic. Table 6.1 shows the false positives for each multiple. The predicted number of packets at which the covert channel would become visible fell between 200 and 410, depending on the hour and packet length. This was calculated using Equation 4.4, the multiple of 6 and the baseline standard deviations for the packet lengths during regular working hours. All injections were conducted during regular working hours.

With the number of packets set, the test consisted of using the steps outlined in Chapter 5 to inject 300 to 1000 packets on each packet length between 1181 and 1190. This was considered an injection set with one injection set conducted every 18 minutes. This allowed for only one injection set for each 17-minute unit of time set in Chapter 5. In total, 19 injection sets were conducted with the results shown in Table 6.2. The high number of false negatives was due to trying to send packets at rates for the 6 standard deviation multiple. Several times the overt traffic would be lower than normal and the channel would not be detected. It was also observed that the collector would drop between 0 to 0.75 percent of packets. These factors made it difficult to inject exactly 6 standard deviations worth of data and required about 30 additional injection sets worth of testing before testable inject values were found. If the inject rate was raised above the 6 standard deviation multiple, all the injected traffic was detected (Table 6.3).

Using Equation 4.4, Table 6.4 shows the predicted bandwidth for a 10 packet length channel between 1181 and 1190 and the actual bandwidth from the testing. The data used the higher data-rate injects used in Table 6.3.

Table 6.1: Determining Ideal Multiple of Standard Deviation

Multiple	False Positives
1	439
2	150
3	47
4	12
5	1
6	0

Table 6.2: Detection of Covert Channel

Channels Injected	Channel Detections	False Positives	False Negatives
110	59	9	29

Table 6.3: Detection of Covert Channel with Higher Inject Rates

Channels Injected	Channel Detections	False Positives	False Negatives
80	80	4	0

Table 6.4: Actual vs. Predicted Bandwidth of 10-Packet Length Channel

	Bandwidth
Predicted	15.550 bps
Actual	17.520 bps

6.3 Detecting Packet Length Covert Channels: Network-Wide

Problems arose once we scaled the narrow testing to all packet lengths. Many of the packet lengths proved extremely volatile making it difficult to compute the standard deviation multiple. When we say volatile here, we refer to packets that have large packet length standard deviation. Some packet lengths required very large multiples in order to remove false positives while others only required small multiples. To fix this, we used different standard deviation multiples depending on the packet length, average packet count and the standard deviation for a given packet. This had the effect of allowing for more variability in volatile packet length regions while maintaining narrow variability in regions with little volatility. Appendix C.2 outlines specific SQL commands used. The multiples varied from 5 to 50 with the most volatile packet lengths getting larger multiples.

Using Equation 4.4, we were able to calculate the maximum undetectable bandwidth for all the packet lengths throughout the day. Figure 6.3 shows the possible channel bandwidths in bits per second for a 20 packet length covert channel.

Even with the variable standard deviation multiples, some false positives remained. However, these false positives rarely exceeded 10 to 20 packet lengths for a 17-minute time interval. Therefore, a value between 10 and 20 was found to be a good number for the alerts variable in Equation 4.6. This would mean a covert channel would need to exceed the threshold function for 10-20 packet lengths in order for it to be flagged as abnormal using our measurement techniques.

Experiments were then conducted by injecting a 20-packet length covert channel using the same methods outlined in the narrow scope section (Section 6.2), but with the standard deviation multiples found in Appendix C.2. Table 6.5 shows the detection statistics for these tests, the actual versus predicted bandwidths.

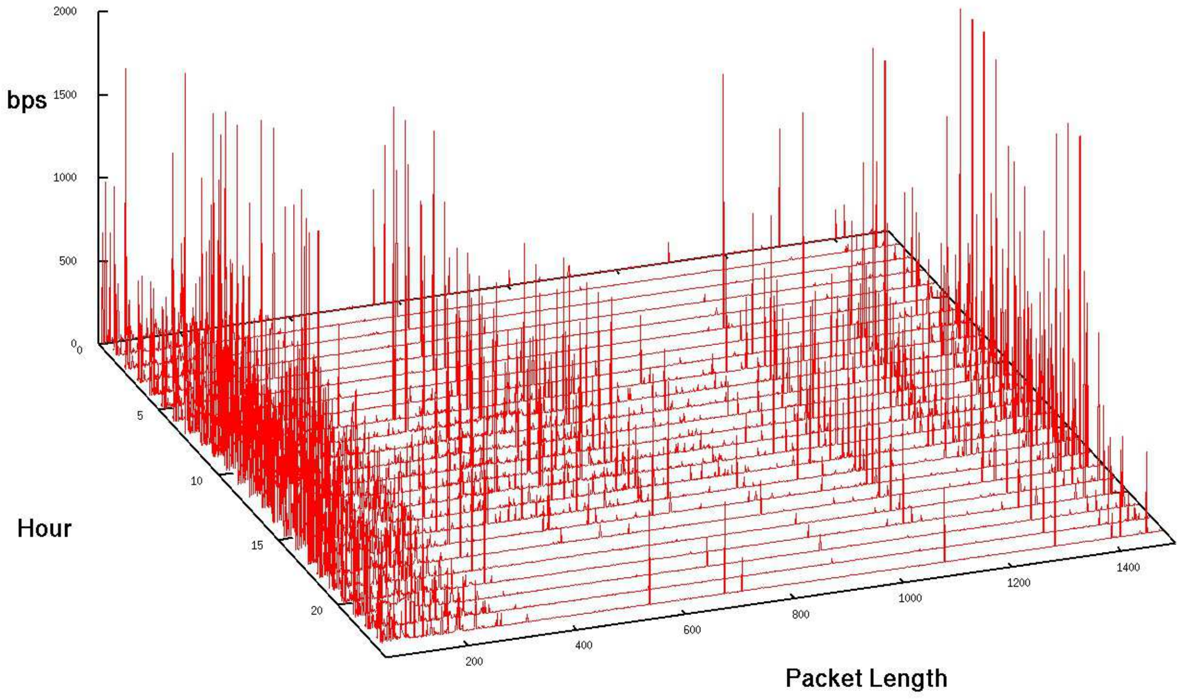


Figure 6.3: Channel Bandwidth for a 20-Packet Length Covert Channel

Table 6.5: Detection of Covert Channel-Network Wide

Channels Injected	Channel Detections	False Positives/Time Unit	False Negatives
160	93	5	67

Table 6.6: Detection of Covert Channel with Higher Inject Rates-Network Wide

Channels Injected	Channel Detections	False Positives/Time Unit	False Negatives
120	118	4	2

Table 6.7: Actual vs. Predicted Bandwidth of 20-Packet Length Channel

	Bandwidth
Predicted	79.986 bps
Actual	86.105 bps

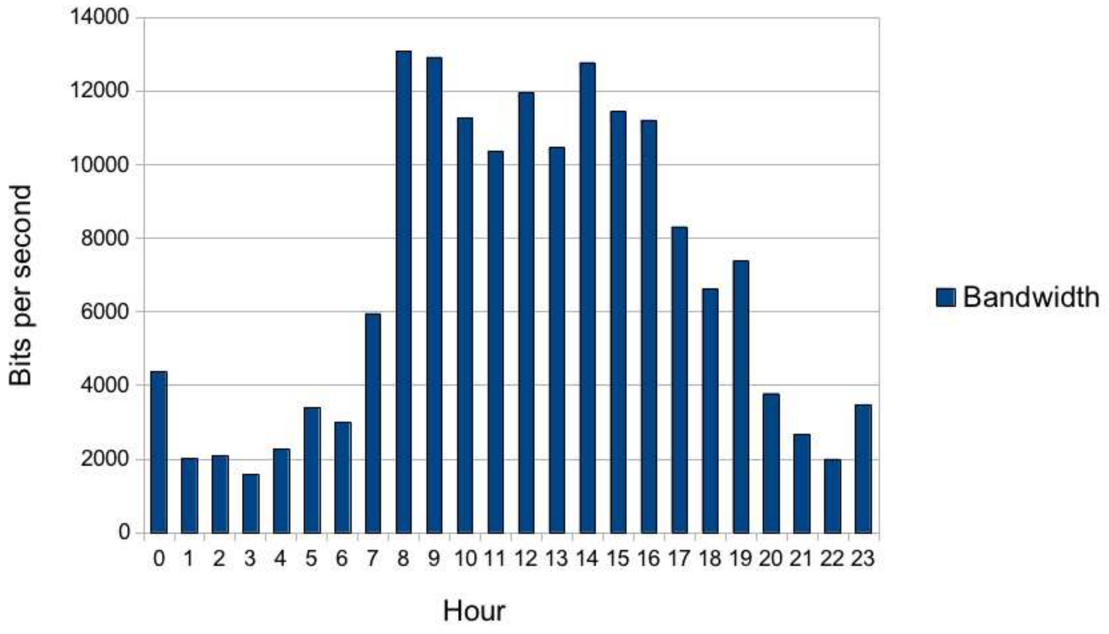


Figure 6.4: Hourly Bandwidth for a 20-Channel Covert Channel

With these numbers, one can now predict possible bandwidths for an undetectable packet based covert channel on this network. Figure 6.4 shows the hourly maximum predicted undetectable bandwidth of a 20 packet length covert channel. This assumes the channel uses the highest bandwidth channels based on Equation 4.4 and the baseline data. It also assumes the messages being sent are evenly distributed, which would cause the user to use the least variable packet length transfer rate for the rest of the packet lengths. Otherwise, the channel would transmit at rates higher than the least variable packet length and would cause detection. The figures would be even higher if the user had a message distribution that perfectly matched the distribution of the traffic. Under these assumptions, some packet lengths such as lengths between 50 and 70 used for TCP acks or packets over 1400 bits, had as high as several kilobits per second bandwidth.

As indicated by Figure 6.4, the channel's peak bandwidth happened during the middle of the day, with a bandwidth of around 13,000 bits per second. This does not seem to be a lot, especially when compared to the large amount of legitimate data that is getting transmitted. But, if one assumes the channel continuously transmits data at rates consistent with Figure 6.4 for an entire day, a total of nearly 600 Mb of data will be transmitted. This is a significant amount of data. Even if one assumes the threshold function could be made more exact, with a resulting reduction in the bandwidth of the channel, its bandwidth would probably still be in the hundreds of megabits per day.

Figure 6.5 shows the bandwidth in megabits per day for channels using between 2 to 1400 packet lengths. This assumes an even distribution for any covert message being sent, which would cause the use of the least variable packet length transfer rate as discussed before. One can immediately see the bandwidth peaks at 800 Mb before gradually dropping down as the number of packet lengths increase. This can be explained by analyzing the variability of the packet lengths. There are only about 100 packet lengths that have a high degree of variability, but after that there is a marked decrease into the lesser-used packet lengths. As the channel becomes wider and wider, it must transmit at slower and slower rates in order to avoid detection on the lesser-used lengths.

If a user had a covert message with the exact distribution of the network or had a method to map the message to the distribution of the network, one could then take advantage of the high variability of the often-used lengths. The bandwidth of this scheme can been seen in Figure 6.6. Here, the bandwidth continues to increase, but the increase slows as the channel uses more and more of the lesser used packet lengths.

These figures illustrate how much data could be exfiltrated without being detected. While these numbers may be high due to the simple threshold function being used, the figure may still be in the several hundred magabits of data per day.

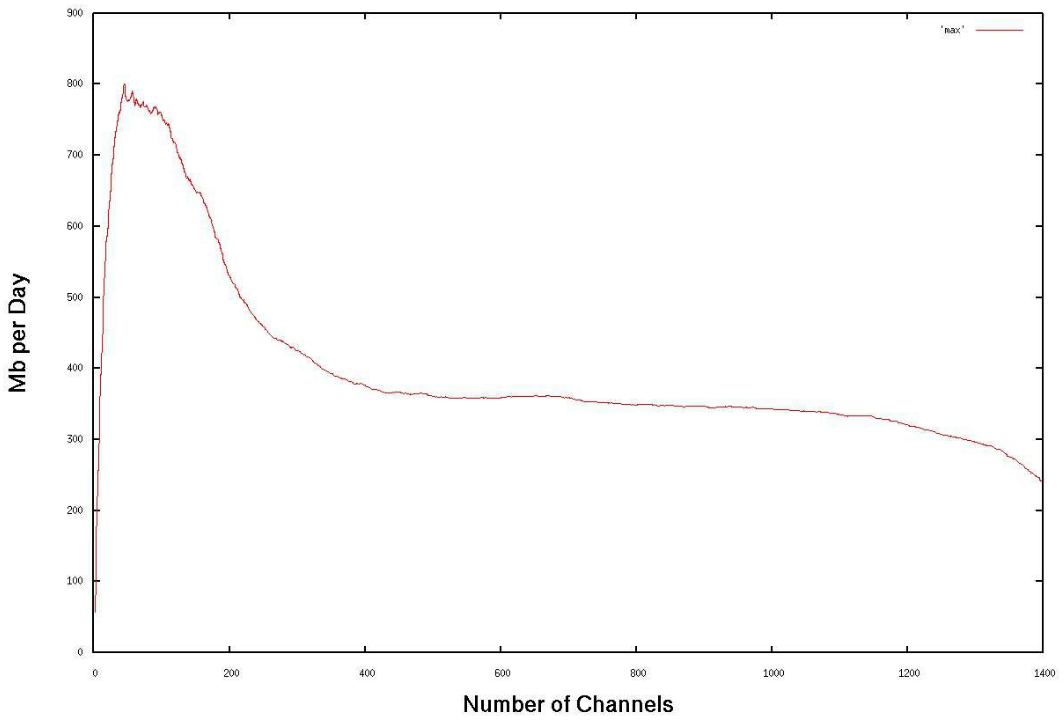


Figure 6.5: Bandwidth vs. Used Packet Lengths with Even Distribution

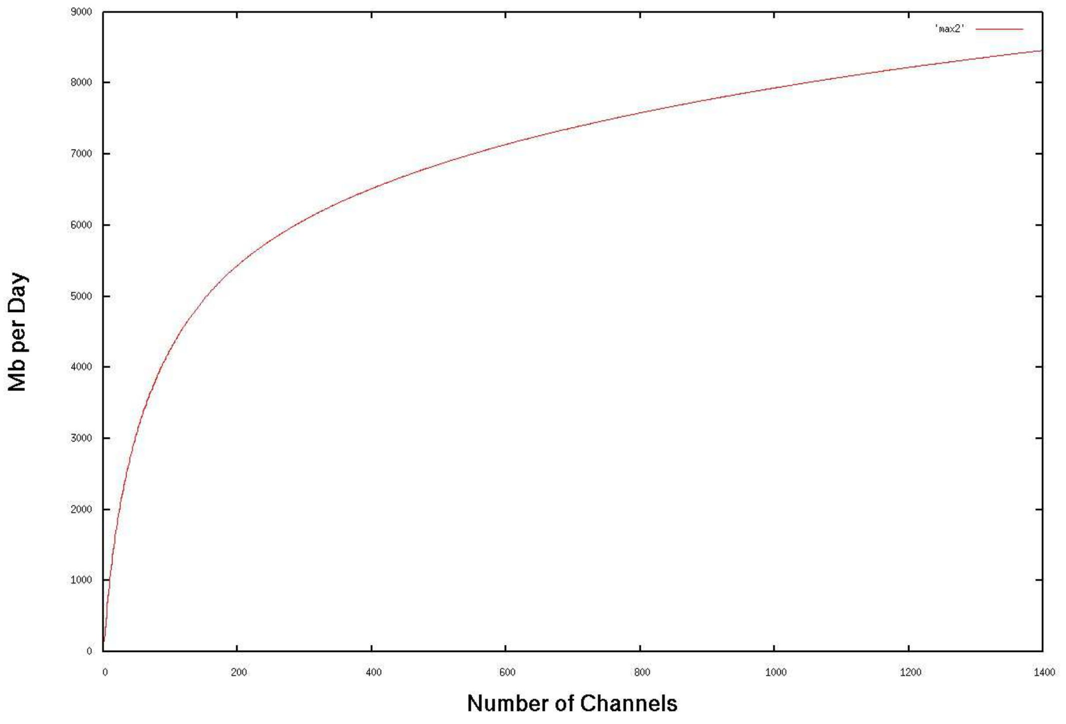


Figure 6.6: Bandwidth vs. Used Packet Lengths

CHAPTER 7:
Conclusion and Further Work

This thesis has shown that it is possible to detect packet length covert channels while monitoring large-scale networks. It has also shown how variable network traffic can be and how it affects the detectability of covert channels. In this thesis, we also demonstrated that it is possible to accurately predict the bandwidth of packet length covert channels and showed that, even at low bandwidths, a covert channel could transmit large amounts of data over the course of a day. The following section outlines some further work that could improve the detectability and robustness of the covert channel.

7.1 Further Work

7.1.1 Better Threshold Function

The threshold function used here relies on basic statistical analysis. Another function that uses more sophisticated mathematical tools could possibly better identify what is considered normal traffic and separate it from unusual traffic. Initial research for this thesis focused on using a DFT calculated for each packet over the course of time. The function proved to be promising, but was only able to detect channels that used packet lengths grouped together or ones that transmitted large amounts of data using a small number of packet lengths. A possible area of research would be to continue investigating the use of DFTs in a threshold function. One idea would be to use the function outlined in reference [18] on a network level instead of simply on a per-connection basis.

The unit of time for the threshold function could also be adjusted. We used 17 minutes due to its speed and efficiency of using bit-wise operations. It would be interesting to test the effectiveness of shortening the time interval or lengthening it. Also, the location of the collector could be adjusted. We placed it fairly high in the network and collected data on an entire department. Further research could test the effectiveness of moving the collection down in the network, even down to a single computer.

7.1.2 Covert Channel Refinement

The channel we defined offers a basic covert channel. We did not need a more robust channel for testing purposes, but several steps can be done if this is needed. First, a better reference can be used. For example, instead of grouping the channel in a single band as in our channel, one could distribute it across the range of packet lengths. One could also use the methods already discussed in [18].

The channel can also be made more robust by expanding the channel into multiple connections to the same peer. This would likely increase the bandwidth of the channel, but would require adding packet tracking and reassembly at the covert channel level.

Interaction between kernel and user-space could also be improved with more robust function calls. Right now, a simple array of buffers is passed between the two. This could be modified to include reading single characters or only part of the receive buffers. Also, a completely kernel-space channel could be developed. Lastly, a fully user-space channel is possible. A user could modify a popular application like Firefox and manually fragment data to desired sizes before passing them down the TCP stack of an unmodified kernel. Further research could be done in this area.

The kernel code could also be better implemented. Currently, the code sometimes introduces artifact packets because of changing the MSS value. They are small packets of less than 400 bytes created from left-over packet data used for the covert channel. Having these artifacts could potentially increase detection by creating possible patterns between packet lengths, or by increasing the occurrence of certain packet lengths. This could be fixed by adding functionality that would remove the small packets and add their data back into the connection's send buffer.

The covert channel code could be implemented in a kernel module. Currently, the channel is hard-coded into the kernel, resulting in an immobile system. An improvement would be to create a kernel module that encompasses the covert channel. This kernel module could then be tied into an existing rootkit.

Lastly, one could try and implement the channel on non-TCP protocols, like UDP. TCP was easy to modify in the kernel and, due to TCP's stateful nature, we did not have to worry about the ordering of the packets as they arrive. UDP may be easier to modify in the kernel, but one would have to build some mechanism into the cover channel to account for packets arriving out of order. This is necessary, since UDP does not track ordering of packets. It also may be worth researching the use of this channel at the IP level. This would be difficult, since the channel would have to be mindful of the protocols higher in the network stack. The channel would have to modify the size of the packet within the TCP or UDP header in addition to crafting a custom

IP packet size. Alternatively, the channel could use IP fragmentation to break up the IP packet without needing to adjust higher level protocols. IP fragmentation allows a sender to break up a message that is too big for the network it wants to send on. This method would probably be easily detectable, however, since IP fragmentation is probably not very common.

REFERENCES

[1] B. W. Lampson, “A note on the confinement problem,” *Commun. ACM*, vol. 16, pp. 613–615, October 1973.

[2] J. Butler, *Rootkits: Subverting the Windows Kernel*. Upper Saddle River: Pearson Education, 2005.

[3] G. J. Simmons, “The prisoners’ problem and the subliminal channel,” in *Proc. CRYPTO’83*, 1983, pp. 51–67.

[4] S. Zander, G. Armitage, and P. Branch, “A survey of covert channels and countermeasures in computer network protocols,” *Communications Surveys Tutorials, IEEE*, vol. 9, no. 3, pp. 44–57, 2007.

[5] J. Liping, J. Wenhao, D. Benyang, and N. Xiamu, “A novel covert channel based on length of messages,” in *Information Engineering and Electronic Commerce, 2009. IEEC ’09. International Symposium*, May 2009, pp. 551–554.

[6] *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, Department of Defense, December 1985.

[7] M. Bishop, *Computer Security: Art and Science*. Westford: Pearson Education, 2003.

[8] R. A. Kemmerer, “Shared resource matrix methodology: an approach to identifying storage and timing channels,” *ACM Trans. Comput. Syst.*, vol. 1, pp. 256–277, August 1983.

[9] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil, “Eliminating steganography in internet traffic with active wardens,” in *Information Hiding*, ser. Lecture Notes in Computer Science, 2003, vol. 2578, pp. 18–35.

[10] T. Handel and M. Sandford, “Hiding data in the OSI network model,” in *Lecture Notes in Computer Science*, 1996, vol. 1174, pp. 23–38.

[11] J. Rutkowska, “The implementation of passive covert channels in the linux kernel,” in *Chaos Communication Congress*, December 2004.

[12] L. Xiapu, E. Chan, and R. Chang, “TCP covert timing channels: Design and detection,” in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 2008, pp. 420–429.

[13] A. Donaldson, J. McHugh, and K. Nyberg, “Covert channels in trusted LANs,” in *Proc. 11th NBS/NCSC National Computer Security Conf*, 1988, pp. 226–232.

[14] M. Handley, V. Paxson, and C. Kreibich, “Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics,” in *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, vol. 10, 2001, pp. 9–9.

[15] M. Padlipsky, D. Snow, and P. Karger, “Limitations of end-to-end encryption in secure computer networks,” Mitre Corp., Tech. Rep. ESD-TR-78-158, 1978.

[16] T. Sohn, J. Seo, and J. Moon, “A study on the covert channel detection of TCP/IP header using support vector machine,” in *Information and Communications Security*, ser. Lecture Notes in Computer Science, 2003, vol. 2836, pp. 313–324.

[17] B. Venkatraman and R. Newman-Wolfe, “Capacity estimation and auditability of network covert channels,” in *IEEE Proceedings on Security and Privacy*, May 1995, pp. 186–198.

[18] A. S. Nair, A. Sur, and S. Nandi, “Detection of packet length based network steganography,” in *International Conference on Multimedia Information Networking and Security*, 2010, pp. 574–578.

[19] C. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, pp. 379–423, 1948.

[20] A. Hero, “Secure space-time communication,” *Information Theory, IEEE Transactions on*, vol. 49, no. 12, pp. 3235–3249, 2003.

[21] (2004) Pcap next generation dump file format. http://www.tcpdump.org/pcap/pcap.html. [Online; accessed 18-January-2011].

[22] (2010) Gnuplot. http://www.gnuplot.info/. [Online; accessed 18-January-2011].

[23] (2006) Hping-active network security tool. http://www.hping.org/. [Online; accessed 18-January-2011].

APPENDIX A: Linux Kernel Modifications

A.1 tcp.c

```

static int cov_channel_rec_index_start=0;
EXPORT_SYMBOL(cov_channel_rec_index_start);
static int cov_channel_rec_index_len=0;
EXPORT_SYMBOL(cov_channel_rec_index_len);

static int cov_channel_send_index_start=0;
EXPORT_SYMBOL(cov_channel_send_index_start);
static int cov_channel_send_index_end=0;
EXPORT_SYMBOL(cov_channel_send_index_end);

static int cov_channel_receive_buf[1000];
EXPORT_SYMBOL(cov_channel_receive_buf);
static int cov_channel_send_buf[1000];
EXPORT_SYMBOL(cov_channel_send_buf);
static int channel_has_contents=0;

int cov_channel_locked=0;
EXPORT_SYMBOL(cov_channel_locked);

uint32_t cov_channel_address=htonl(INADDR_LOOPBACK);
EXPORT_SYMBOL(cov_channel_address);

asmlinkage long sys_cov_channel_read(char * buffer, int* len)
{
   printk(KERN_EMERG "sys_cov_channel_read length %d",
       cov_channel_rec_index_len);

        if(cov_channel_rec_index_len==0)
                return -1;

        long err=0;

        long temp_buff[1000];
        int x=0;
        for(x=0;x<cov_channel_rec_index_len;x++)
        {
                temp_buff[x]=cov_channel_receive_buf[x];
                cov_channel_receive_buf[x]='\0';
        }

        temp_buff[x]='\0';
        x=0;
        while(x<cov_channel_rec_index_len)
        {
                buffer[x]=temp_buff[x];
                x++;
        }
        buffer[x]='\0';
        *len= cov_channel_rec_index_len;
        err=put_user(cov_channel_rec_index_len,len);

        cov_channel_rec_index_len=0;
        cov_channel_rec_index_start=0;

  printk(KERN_EMERG "hello world!%s",buffer);
  return err;
}

asmlinkage long sys_cov_channel_write(char* buffer)
{
        long err=0;
        if ( (strlen(buffer) >= ( (cov_channel_send_index_start -
            cov_channel_send_index_end + 1000) % 1000)) &&
            channel_has_contents )
                return -1;

        int x=0;
        int buffer_x=cov_channel_send_index_end;
        for(x=0; x< strlen(buffer); x++)
        {
                cov_channel_send_buf[buffer_x]=buffer[x];
                if(buffer_x+1==1000)
                        buffer_x=0;
                else
```

```
                        buffer_x++;
        }

        cov_channel_send_index_end= (cov_channel_send_index_end + strlen(
            buffer)) % 1000  ;
        channel_has_contents=1;

  printk(KERN_EMERG "The world hello ");

  return err;
}

asmlinkage void sys_cov_channel_peer(long add)
{
        cov_channel_address=add;
}
void channel_put_package_received(int package)
{
  printk(KERN_EMERG "channel_put_package_reveived -%d",package);

        if(cov_channel_rec_index_len==1000)
                return;
        cov_channel_receive_buf[cov_channel_rec_index_len]=package;
        cov_channel_rec_index_len++;
}

int channel_get_package_to_send(void)
{
        if(!channel_has_contents)
                return -1;
        int packet=cov_channel_send_buf[cov_channel_send_index_start];

        cov_channel_send_buf[cov_channel_send_index_start]=0;
        cov_channel_send_index_start=(cov_channel_send_index_start+1) %
            1000;
        if(cov_channel_send_index_start==cov_channel_send_index_end)
                channel_has_contents=0;

        return packet;
}

```

```
int channel_packet_check()
{
        if(!channel_has_contents)
                return -1;
        return cov_channel_send_buf[cov_channel_send_index_start];
}

EXPORT_SYMBOL(channel_put_package_received);
EXPORT_SYMBOL(channel_get_package_to_send);
EXPORT_SYMBOL(channel_packet_check);
```

A.2 tcp_input.c

```
extern void channel_put_package_received(int package);
extern int cov_channel_address;

static void tcp_event_data_recv(struct sock *sk, struct sk_buff *skb)
{
        /*check if the packet is coming from the cov host and if it's
            between the packet window*/
        /* This is the only mod for receive */
        if(inet_sk(sk)->daddr==cov_channel_address && skb->len >= 1202 &&
            skb->len <=1400)
                channel_put_package_received(skb->len-1202);

        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        u32 now;

        inet_csk_schedule_ack(sk);

        tcp_measure_rcv_mss(sk, skb);

        tcp_rcv_rtt_measure(tp);

        now = tcp_time_stamp;

        if (!icsk->icsk_ack.ato) {
                /* The _first_ data packet received, initialize
                 * delayed ACK engine.
                 */
                tcp_incr_quickack(sk);
                icsk->icsk_ack.ato = TCP_ATO_MIN;
        } else {
                int m = now - icsk->icsk_ack.lrcvtime;

                if (m <= TCP_ATO_MIN / 2) {
                        /* The fastest case is the first. */
                        icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) +
                            TCP_ATO_MIN / 2;
                } else if (m < icsk->icsk_ack.ato) {
                        icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + m;
                        if (icsk->icsk_ack.ato > icsk->icsk_rto)
                                icsk->icsk_ack.ato = icsk->icsk_rto;
                } else if (m > icsk->icsk_rto) {
                        /* Too long gap. Apparently sender failed to
                         * restart window, so that we send ACKs quickly.
                         */
                        tcp_incr_quickack(sk);
                        sk_mem_reclaim(sk);
                }
        }
        icsk->icsk_ack.lrcvtime = now;

        TCP_ECN_check_ce(tp, skb);

        if (skb->len >= 128)
                tcp_grow_window(sk, skb);
}
```

A.3 tcp_output.c

```

/* This routine writes packets to the network.  It advances the
 * send_head.  This happens as incoming acks open up the remote
 * window for us.
 *
 * LARGESEND note: !tcp_urg_mode is overkill, only frames between
 * snd_up-64k-mss .. snd_up cannot be large. However, taking into
 * account rare use of URG, this is not a big flaw.
 *
 * Returns 1, if no segments are in flight and we have queued segments, but
 * cannot send anything now because of SWS or another problem.
 */
extern int channel_get_package_to_send();
extern int channel_packet_check();
extern int cov_channel_locked;
extern int cov_channel_address;

static int tcp_write_xmit(struct sock *sk, unsigned int mss_now, int
    nonagle,
                          int push_one, gfp_t gfp)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        unsigned int tso_segs, sent_pkts;
        int cwnd_quota;
        int result;

        struct inet_sock *inet = inet_sk(sk);

        sent_pkts = 0;

        /* lock the channel if not already locked */
        if(!sk->is_cov_channel && !cov_channel_locked && inet->daddr==
            cov_channel_address)
        {
                cov_channel_locked=1;
                sk->is_cov_channel=1;
        printk(KERN_EMERG "###Channel locked!!");

        }

        if (!push_one) {
                /* Do MTU probing. */
                result = tcp_mtu_probe(sk);
                if (!result) {
                        return 0;
                } else if (result > 0) {
                        sent_pkts = 1;
                }
        }

        int real_mss=mss_now;
        int packet=-1;
        int x=0;
        while ((skb = tcp_send_head(sk))) {
                x++;
                packet=channel_packet_check();
                int packet_used=0;

                /*get a value to send and format the mss value accordingly
                    */
                if(sk->is_cov_channel && packet!=-1 && skb->len > (1202+
                    packet))
                {
                        packet_used=1;
                        printk(KERN_EMERG "package_to_send␣-%c␣-%d",packet,
                            packet);
                        mss_now = 1202 + packet;
                }
                else if(skb->len >= 1202 && skb->len <=1400)
                        mss_now=1200;
                else
                        mss_now= real_mss;


                unsigned int limit;

                tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
                BUG_ON(!tso_segs);

                cwnd_quota = tcp_cwnd_test(tp, skb);
                if (!cwnd_quota)
                        break;

                if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
                        break;

                if (tso_segs == 1) {
                        if (unlikely(!tcp_nagle_test(tp, skb, mss_now,
                                                     (tcp_skb_is_last(sk,
                                                         skb) ?
                                                      nonagle :
```

```
                                                TCP_NAGLE_PUSH))))
                                break;
                } else {
                        if (packet==-1 && !push_one && tcp_tso_should_defer(sk, skb))
                                break;
                }

                limit = mss_now;
                if (tso_segs > 1 && !tcp_urg_mode(tp))
                        limit = tcp_mss_split_point(sk, skb, mss_now,
                                                    cwnd_quota);
                if(packet!=-1)
                        limit=mss_now;

                if (skb->len > limit &&
                    unlikely(tso_fragment(sk, skb, limit, mss_now)))
                                break;
                TCP_SKB_CB(skb)->when = tcp_time_stamp;

                /* remove from buffer if the packet was sent with a covert value */
                if(packet_used)
                        packet=channel_get_package_to_send();
                if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
                        break;

                /* Advance the send_head.  This one is sent out.
                 * This call will increment packets_out.
                 */
                tcp_event_new_data_sent(sk, skb);

                tcp_minshall_update(tp, mss_now, skb);
                sent_pkts++;

                if (push_one)
                        break;
        }

        if (likely(sent_pkts)) {
                tcp_cwnd_validate(sk);
                return 0;
        }
        return !tp->packets_out && tcp_send_head(sk);
}
```

A.4 reader.c

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <string.h>

long channel_read(char* buffer, int* len)
{
  return syscall(337,buffer,len);
}

long channel_write(char* buffer)
{
  return syscall(338, buffer, strlen(buffer));
}

long channel_set_peer(long add)
{
  return syscall(339, add);
}

int main()
{
  channel_set_peer(100837568);
  int x=0;
  char name [] = "Hidden in plain sight";
  char buffer[1000];
  int len=0;
  long package= channel_write(name);
  long err= channel_read(buffer,&len);
  printf("from the network- %s\n",buffer);

  return 1;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Collection Code

B.1 PCAP Packet Length Extractor

```
#include <stdio.h>
#include <pcap.h>
#include <stdlib.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#define ETHER_TYPE_IP (0x0800)
#define ETHER_TYPE_8021Q (0x8100)
#define OFFSET  (0xXXXFFF)
#define NETMASK (0xXXXXX)

int main(int argc, char **argv)
{
  struct pcap_pkthdr header;
  const u_char *packet;

  int y=0;
  char src_ip[100], dst_ip[100];

  pcap_t *handle;
  char error[PCAP_ERRBUF_SIZE];
  handle = pcap_open_offline(argv[y], error);
  int packet_length;

   while (packet = pcap_next(handle,&header)) {
    u_char *packet = (u_char *)packet;

    int ethernet_type = ((int)(packet[12]) << 8) | (int)packet[13];

    if (ethernet_type == ETHER_TYPE_IP)
    {
        packet += 14;
        struct ip *ip_hdr = (struct ip *)packet;
        struct iphdr *ip_header = (struct ip *)packet;
        
        packet_length = ntohs(ip_hdr->ip_len);
        
        if(packet_length> 55 && packet_length < 1500 &&
           ((ip_header->saddr&OFFSET)==NETMASK ||
           (ip_header->daddr&OFFSET)==NETMASK))
        {
           printf("%d_%d\n",header.ts.tv_sec, packet_length);
        }
      }
    }
    pcap_close(handle);
  return 0;
}
```

B.2 Network Packet Reader

```
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#define ETHER_TYPE_IP (0x0800)
#define ETHER_TYPE_8021Q (0x8100)
#define OFFSET   (0xF8FFFF)
//#define NETMASK  (0xA8C0)
#define NETMASK (0x6814AC)

void callback(u_char *useless,const struct pcap_pkthdr* header,const u_char
    *
        pkt_ptr)
{
    if ( (((int)(pkt_ptr[12]) << 8) | (int)pkt_ptr[13]) == ETHER_TYPE_IP)
    {
          pkt_ptr += 14;  //skip past the Ethernet II header
          struct ip *ip_hdr = (struct ip *)pkt_ptr; //point to an IP header
               structure
        struct iphdr *ip_header = (struct ip *)pkt_ptr; //point to an IP
            header structure

        int packet_length = ntohs(ip_hdr->ip_len);

        if(packet_length> 55 && packet_length < 1500 &&
            ((ip_header->saddr&OFFSET)==NETMASK ||
             (ip_header->daddr&OFFSET)==NETMASK))
        {
            printf("%d %d\n",header->ts.tv_sec, packet_length);
        }
    }
}

int main(int argc,char **argv)
{
    char err[PCAP_ERRBUF_SIZE];

    descr = pcap_open_live("eth0",100,0,-1,errbuf);
    if(descr == NULL)
    {
        printf("pcap_open_live(): %s\n",err); exit(1);
    }

    pcap_loop(descr,-1,callback,NULL);

    return 0;
}
```

B.3 Batch Collection Files to MySQL loader

```
#!/bin/bash

front="load data infile '/home/derek/Documents/NPS/thesis/pcap\ data/oct20/
    "
back="' into table rawdata FIELDS TERMINATED BY ' ' IGNORE 1 LINES; ";

for X in *txt
do
    echo $front$X$back
done
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: MySQL Table Definitions

C.1 Table Definitions

```
-- MySQL dump 10.13  Distrib 5.1.52, for pc-linux-gnu (x86_64)
--
-- Host: localhost    Database: paclen
-- ------------------------------------------------------
-- Server version	5.1.52-log

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
    FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `ave`
--

/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `ave` (
  `len` int(11) NOT NULL,
  `ave` bigint(20) unsigned NOT NULL,
  `stdev` bigint(20) unsigned NOT NULL,
  `hour` int(11) NOT NULL,
  PRIMARY KEY (`len`,`hour`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
```

```
-- Table structure for table `count`
--

/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `count` (
  `timestamp` bigint(20) unsigned NOT NULL,
  `len` int(11) NOT NULL,
  `counter` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`timestamp`,`len`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;


--
-- Table structure for table `rawdata`
--

/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `rawdata` (
  `timestamp` bigint(20) unsigned NOT NULL,
  `len` int(11) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;
```

C.2 Setting Standard Deviation Multiple

```
update ave3 set c_std=1 where max > 2000000;
update ave3 set c_std=2 where max > 400000 and c_std=0;
update ave3 set c_std=5 where max > 100000 and c_std=0;
update ave3 set c_std=8 where max > 50000 and c_std=0;
update ave3 set c_std=30 where stdev < 50 and ave < 150 and c_std=0;
update ave3 set c_std=25 where stdev > 1000 and ave > 2000 and c_std=0;
update ave3 set c_std=30 where (len < 300 or len > 1399) and c_std=0;
update ave3 set c_std=25 where ave > 200 and len > 600 and len < 900 and
    c_std=0;
update ave3 set c_std=25 where ave > 250 and len > 400 and len < 600 and
    c_std=0;
update ave3 set c_std=20 where ave > 500 and len > 1150 and len < 1300 and
    c_std=0;
```

```
update ave3 set c_std=10 where c_std=0;
```

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. George Dinolt
   Naval Postgraduate School
   Monterey, California

4. Bret Michael
   Naval Postgraduate School
   Monterey, California

5. Chris Eagle
   Naval Postgraduate School
   Monterey, California

6. John McEachen
   Naval Postgraduate School
   Monterey, California

7. Hersch Loomis
   Naval Postgraduate School
   Monterey, California

8. ITACS
   Naval Postgraduate School
   Monterey, California